Dustin Horvath
2729265
EECS 665 Lab 09

The purpose of this lab is to take intermediate code and produce actionable assembly-level code using Lex and Yacc. A good portion of this code is provided in the files *cgen.l* and *cgen.y*.

The file *cgen.l* contains lexical rules that tokenize the intermediate code within the file *test.t*. This is necessary for Yacc to be able to interpret the text and apply it's grammar rules for parsing. Luckily, the intermediate code language is relatively simple by design, so the rules needed for tokenizing are straightforward and the list of rules is not long.

The entirety of this program contains a number of auxiliary files such as *arg.**, *function.**, *local.**, and *strng.**. These files contain functionality for things like string handling, and the naming and declaration of functions. These files are generally necessary for the program to run at all and produce output, and have been provided for us entirely.

The bulk of configuration for proper output is done through changes within the file *cgen.y*. This file contains the grammar rules for parsing the intermediate code within *test.t*, and is used by Yacc to produce our assembly output. The bulk of this code was given to us, and provided a suitable framework for completing the components needed for the remaining functionality.

Additional rules were defined for the functions "call", "return", "argument", arithmetic shifts, and some arithmetic operations. Example operations already in place helped avoid trial and error and helped avoid pruning through code to understand the function of particular variables.

For the arithmetic functions, basic assembly is produced in the standard manner. Necessary arguments for function calls are pushed on the stack, frame pointers are saved

where needed, stack space is allocated by using ‘subl $amountofspace, %esp’. The register %eax is used for retrieving operators for a particular function, and points to the location of the first operator in memory; subsequent operators can be retrieved with an offset. Any needed temporary arguments are created using the ‘movl’ command and a memory location relative to the stack pointer; this allows us to create space for any temporary values wherever needed. Ultimately, the result of a function being parsed and output as assembly code will have the output being stored in the %eax register. The stack pointer is then adjusted back to the correct location, and the function *ret*urns to the previous line of instruction code from before it was called.

A good deal of this lab included matching the correct format of output so that the resulting assembly code would be valid. This includes correctly indenting lines and producing assembly output in the correct order. This work was somewhat tedious but not difficult.

The most challenging aspects of the lab were correctly handling the variables output by Yacc, and properly addressing the register values used for storing memory locations. Some functions required looping over the number of arguments *$3* output by Yacc, and all required getting the values themselves with *$2*. Using the correct arguments was also a bit tedious, and (at least for myself) incredibly prone to programmer error.